Tackling High Variability in Video Surveillance Systems through a Model Transformation Approach

Mathieu Acher, Philippe Lahire, Sabine Moisan, Jean-Paul Rigault

Tackling High Variability in Video Surveillance Systems through a Model Transformation Approach

Mathieu Acher, Philippe Lahire
CNRS I3S
Route des Colles
06903 Sophia Antipolis Cedex, France
{acher,lahire}@i3s.unice.fr

Sabine Moisan, Jean-Paul Rigault
INRIA Sophia Antipolis Méditerrannée
Route des Lucioles
06902 Sophia Antipolis Cedex, France
{moisan,jpr}@sophia.inria.fr

Abstract

This work explores how model-driven engineering techniques can support the configuration of systems in domains presenting multiple variability factors. Video surveillance is a good candidate for which we have an extensive experience. Ultimately, we wish to automatically generate a software component assembly from an application specification, using model to model transformations. The challenge is to cope with variability both at the specification and at the implementation levels. Our approach advocates a clear separation of concerns. More precisely, we propose two feature models, one for task specification and the other for software components. The first model can be transformed into one or several valid component configurations through step-wise specialization. This paper outlines our approach, focusing on the two feature models and their relations. We particularly insist on variability and constraint modeling in order to achieve the mapping from domain variability to software variability through model transformations.

1. Introduction

This work explores a possible synergy between the video surveillance software domain and model-driven engineering (MDE). Building video surveillance software is a complex process, with many design decisions, both at specification and implementation levels. On the one hand, we expect that MDE techniques will promote new paradigms in video surveillance design processes. On the other hand, confronting MDE with such a large scale application will certainly raise new challenging problems in MDE itself.

In the video surveillance community, the focus has moved from individual vision algorithms to integrated and generic software platforms, and now to the security, scalability, evolution, and ease of use of these platforms. The last trends require a modeling effort, for video surveillance component frameworks as well as for task specification. Thus this domain is a good candidate not only to put MDE to the test, but also to improve and enrich it.

MDE seems mature enough to be confronted with real applications. A crucial question is to determine the current limits regarding complexity, scalability, and variability issues. The latter item is of major importance and is central to our work. Indeed, we know that a huge number of correct configurations is possible, even with few variation points. This is difficult to master, even by experienced users [9, 12].

Our approach is to apply modeling techniques to the specification (describing the video surveillance task and its context) as well as to the implementation (assembling the software components). The final objective is to define methods mature enough to specify a video surveillance task and its context and to obtain (semi-)automatically a set of valid component assemblies, through model transformations. The key issue is variability representation and management. In this paper we focus on *static* configuration; run-time adaptation and control are a matter for other techniques that we will briefly mention in the conclusion.

In the next section, we introduce the challenges faced by designers of video surveillance applications. The major problems are due to the huge number of variants. Our approach uses feature models to cope with this variability, together with model to model transformations presented in section 3. Section 4 describes our approach, its specific models and their transformations. Finally, we conclude with the current status of the project and its intended evolution.

2. Video Surveillance Processing Chains

The purpose of video surveillance is to process one or several image sequences to detect interesting situations or events. Depending on the application, the corresponding video analysis results may be stored for future processing or may raise alerts to human observers.

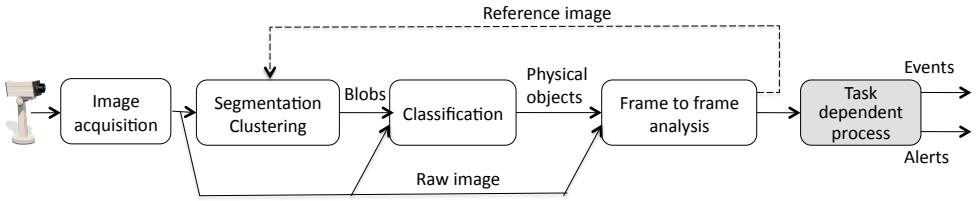


Figure 1. A simplified video surveillance processing chain

There are several kinds of *video surveillance tasks* according to the situations to be recognized: detecting intrusion, counting objects or events, tracking people, animals or vehicle, recognizing specific scenarios... Apart from these functional characteristics, a video surveillance task also sports non functional properties, such as quality of service. The most typical criteria concern the robustness characterized by the number of false positive and negative detections, the response time, the accuracy of object and event recognition... As a matter of examples, intrusion detection may accept some false positives, especially when human operators are monitoring the system; counting requires a precise object classification; recognition of dangerous behavior must be performed within a short delay.

Moreover, each kind of task has to be executed in a particular context. This context includes many different elements: information on the objects to recognize (size, color, texture...), description and topography of the scene under surveillance, nature and position of the sensors (especially video cameras), lighting conditions... These elements may be related together, e.g., an indoor scene implies a particular lighting. They are also loosely related to the task to perform since different contexts are possible for the same functionality. For instance, intrusion detection may concern people entering a warehouse or pests landing on crop leaves.

The number of different tasks, the complexity of contextual information, and the relationships among these items induce a high variability at the specification level. Defining several product lines is a usual way to reduce variability. However, each line (intrusion, counting, scenario recognition...) still contains many variability factors.

The variability even increases when considering implementation issues. A typical video surveillance processing chain (figure 1) starts with image acquisition, then segmentation of the acquired images, clustering to group image regions into blobs, classification of possible objects, and tracking these objects from one frame to the other. The final steps depend on the precise task. Additional steps may be introduced, such as reference image updating (if the segmentation step needs it), data fusion (in case of multiple cameras) or even scenario recognition. This pipe-line architecture is rather stable across tasks. By contrast, for each step, many variants exist along different dimensions. For instance, there are various classification algorithms with different ranges of parameters, using different geometrical models of physical objects to recognize, with different strategies to merge and split image blobs to label them as objects. And it is of course the same for other algorithms.

These steps and their variants constitute software components that designers must correctly assemble to obtain a processing chain. For this, current research in video surveillance not only focuses on individual algorithms but also on integrated component frameworks (or platforms) that cover all the steps. Such a framework is being developed [2] in our group: written in C++, it targets most of video surveillance tasks and proposes a choice of algorithms for each step. Such frameworks indeed favor software reuse and assembly safeness; however, they are still delicate to master, a real challenge.

To sum up, designing a video surveillance system requires to cope with multiple sources of variability, both on the task specification side and on the implementation one. For this, following modern software engineering practices, we propose to model both sides, each with its variabilities. Model transformations will then assist the process of producing a video surveillance system from a requirement specification. Separation of concerns makes models easier to manage. However, since concerns are usually not completely independent, we also need to introduce constraints enforcing the relationships inside as well as across models.

3. Variability and Model Transformation

Software video surveillance products exhibit similar features; hence, they can be considered as a software product line (SPL), a.k.a. a product family [13]. A crucial issue is to make explicit their differences in terms of provided features, fulfilled requirements, or execution assumptions. Central to SPL approaches is the ability to deal with product *variability* that is "the ability of a system to be efficiently extended, changed, customized or configured for use in a particular context" [15]. Consequently, we need methods for representing variability, and for efficiently operating on it at each stage of software development. Moreover, model transformations require mechanisms to handle relations between variability models at different abstraction levels.

3.1. Variability Modeling

Modeling variability has been explored in several domains. One of the most practical techniques is *feature modeling*. It aims at modeling the common and variable *features* of a product family. Several definitions of the notion of feature appear in the literature, ranging from "anything user or client programs might want to control about a concept" [4], "a prominent or distinctive user-visible aspect, quality or characteristic of a software system" [6] to "an increment in product functionality" [3]. Feature modeling is not only relevant to requirement engineering but it can also apply at design or implementation time. Furthermore, features are ideal abstractions that customers, experts and developers can easily understand.

The FODA (Feature-Oriented Domain Analysis) method [6] was the first to propose to capture feature models as diagrams. A feature diagram is a set of features, hierarchically organized. Features are nodes of the corresponding tree and can be mandatory or optional. Edges are used to progressively decompose features into sub-features that detail the parent ones. Aggregation and alternative groups (AND and XOR) are examples of such decompositions. In addition, aside from the tree structure, composition rules express dependencies and capture possible complex interactions between features. For instance, it is possible to express that one feature *requires* an other one or that two features are *mutually exclusive* even if they belong to distant sub-trees. As an extension of FODA, the FORM method provides additional types of constraints [7].

A feature model represents a set of *configurations*, each being a set of features consistent with the constraints and the semantics of feature models. As proposed in [5], the process of deriving a valid configuration may be performed in stages, where each stage *specializes* the feature diagram. A feature model f' specializes another one, say f, if the set of configurations represented by f' is a subset of the configurations represented by f. Hence, specialization reduces the set of possible configurations of f by selecting or removing features. Note that a fully specialized feature diagram represents a single configuration.

Since the original definition [6], a plethora of notations and extensions have been proposed [7, 5, 15]. Schobbens *et al.* provide a generic formal definition of feature diagrams that is a generalization of all the variants of feature diagrams [14]. They define a pivot abstract syntax called *Free Feature Diagram*, that constitutes a meta-model of feature diagrams and allows to reason formally on the syntax and semantics of these diagrams.

In our case we have multiple variability factors, both for specifying an application and for describing the software framework. Indeed, it is difficult to reason directly on implementation-oriented abstractions and to integrate requirement level knowledge at the same time. We thus decided to separate these two concerns. This led to two feature models: one for video surveillance *task specification*, the other for *framework description*. Both models address the static configuration phase. This phase raises interesting enough problems that must be solved before tackling run-time control and adaptation that require other kinds of models.

3.2. Model Transformation

The two feature models provide a basis to understand the two aspects of software in video surveillance, the application requirement and the processing chain. The challenging task here is to map the domain variability (or problem space) to software variability (or solution space).

Model-driven engineering uses models to represent partial or complete views of an application or a domain, possibly at different abstraction levels. MDE offers techniques to transform a source model into a target one. Source and target models can reside at the same abstraction level (e.g. specialization of a feature model) or at different abstraction levels (e.g. mapping between task and framework models). A set of transformation rules describes how one or more constructs in the source model can be transformed into one or more constructs in the target one [8].

Our approach combines SPL and MDE. SPL exploits the knowledge of problems in a particular domain and tries to automate the construction of applications. MDE techniques narrow the gap between the problem and solution spaces through transformations. Moreover, MDE puts forward multi-stage strategies to derive software assets by step-wise refinement. MDE and SPL are complementary technologies [12, 19] and model transformations are at the heart of their potential synergy.

4. A MDE Approach to Video Surveillance

4.1. Outline of the Approach

As already mentioned we propose two feature models: a generic model of video surveillance applications (task specification model, for short *task model*) and a model of video processing components and chains (component framework configuration model, for short *framework model*). Both of them are feature models expressing configuration variability factors. The task model describes the relevant concepts and features from the stakeholders' point of view, in a way that is natural in the video surveillance domain: characteristics and position of sensors, context of use (day/night, in/outdoors, target task)... It also includes notions of quality of service (performance, response time, detection robustness, configuration cost...). The framework model describes

the different software components and their assembly constraints (ordering, alternative algorithms...).

It is convenient to use the same kind of models on both sides, leading to a uniform syntax. Feature models are appropriate to describe static variants; they are simple enough for video surveillance experts to express their requirements. Nevertheless, other types of models may deserve consideration (e.g., component models on the framework side) or may be needed when it comes to run-time control (e.g., work-flow models). In all cases, these new models must be compatible with the variability configuration models.

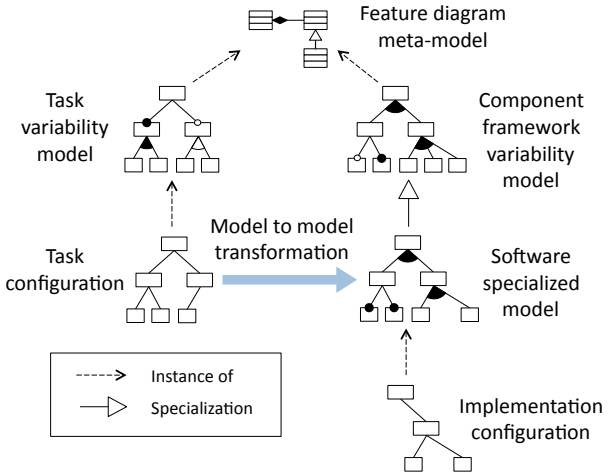


Figure 2. MDE approach to video surveillance

We clearly need to model transformations from the first model to the second, to allow to automatically (or semi-automatically) turn an application specification into a suitable processing chain. Our approach is represented on figure 2. A designer of video surveillance systems instantiates the task model, thus producing a task specification. This latter is a configuration of the task model, consistent with the constraints. Then, the designer triggers the automatic transformation. Due to the multiple causes of variability, the result of the transformation is usually not one single instance of the framework model, but rather a set of possible component assemblies fulfilling the task and context specification. This means that the designer obtains a sub-model, in fact a *specialization* (see 3.1), of the framework model. This sub-model is, by construction, consistent with the constraints in both models. The final configuration of the video processing chain has to be manually fined-tuned by the designer, leading to an instance of the framework model.

Since we wish to define the static configuration of the system, the designer has to choose all the features that would induce the selection of those components that *may* be useful at run-time. Some of these features are imposed by the task definition, other will be controlled at run-time.

4.2. Task Specification and Framework Models

Figure 3(a) shows an excerpt of the feature diagram corresponding to the task specification. To enforce separation of concerns, we identified four top level features. The Task feature expresses the precise function to perform. QoS corresponds to non-functional requirements, especially those related to quality of service. Then, we need to define the Object(s) of interest to be detected, together with their properties. Finally, Scene context is the feature with the largest sub-tree; it describes the scene itself (its topography, the nature and location of sensors) and many other environmental properties (only few of them are shown on the figure).

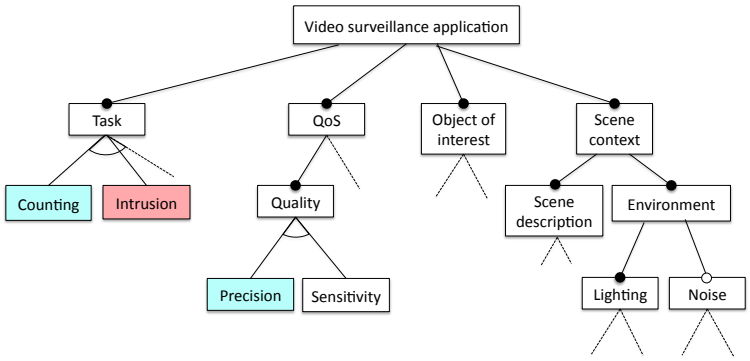
In this diagram, the (sub-)features are not independent. A decision in the task model (e.g., selecting or removing a feature) may impact both the task model itself and the framework model. The corresponding constraint propagation reduces the set of possible configurations both in the problem and in the solution space. Thus, we have enriched the feature diagrams by adding internal constraints to cope with relations local to a model. Constraints across models are related to model transformations (see next section).

So far, we have identified three kinds of constraints. Choosing one feature may **imply** or **exclude** to select an other specific one. In other cases, the choice of one feature only **suggests** to use an other one; this corresponds to default cases. For instance, if Counting is the current task, it **implies** a high detection precision to accurately recognize the objects to count. The corresponding features appear as blue in figure 3(a). By contrast, intruders do not need to be precisely characterized; thus Intrusion **suggests** low precision (but could cope with a high one).

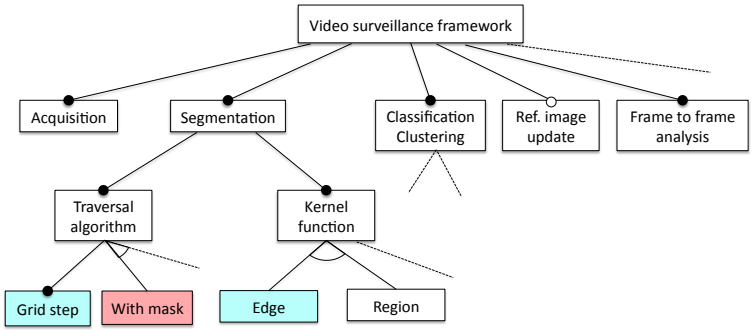
On the framework side, we also used feature diagrams. Figure 3(b) displays a highly simplified form of the corresponding diagram. The top level features mainly correspond to the different steps of the processing pipeline (some of them optional). The figure displays only a few sub-features of the segmentation step. Similarly to the previous diagram, we also introduce internal constraints. They have the same form as before. For instance, edge segmentation **implies** a thin image discretization, thus a low Grid step. The corresponding features are filled in blue on figure 3(b).

4.3. Transformations

Many transformations can be represented as rules of the same form as the previous constraints, but they relate features *across* models. For example, Intrusion **implies** the existence of particular zones of interest such as doors. This is an internal constraint which involves a Context feature (not shown on figure 3(a)). The existence of such zones in turn **suggests** to analyze only the matching parts of the images, which is done by using masks during Segmentation. The



(a) Specification of tasks



(b) Software framework

Figure 3. Simplified feature models: video surveillance task specification and software framework

corresponding features on both diagrams are marked in red. As an other example, the need for high precision in Counting **implies** to use a low Grid step during Segmentation.

We wish to use similar formalisms for internal constraints and transformation rules. Presently, both are written as text. We still have to decide on a formalism which should be expressive enough and preferably verifiable. OCL or systems like ALLOY are worth exploring. Moreover, transformation rules must be executable, which suggests to think of transformation engines (imperative ones such as KERMETA or more declarative like ATL) or even inference rule engines as in Artificial Intelligence. An other alternative is to use a constraint solver to derive a correct and optimal configuration. It makes it possible to specify more global constraints and can be efficient in some situations, although the drawback is an increased complexity. Moreover, means to efficiently translate specification rules into concrete features are still an open issue [18]. In all cases, meta-models of transformations are to be elaborated.

We suspect that the ideal representation could be a mixture of imperative and rule-based techniques. In a previous work, we proposed artificial intelligence techniques, called *Program Supervision*, to control the scheduling and the execution of vision chains [17]. This experience could be adapted to drive model transformations.

5. Conclusion and Future Work

In video surveillance, task requirements as well as software component assembly are complex to handle. The challenge is to cope with the many—functional as well as non-functional—causes of variability on both sides. Hence, we first decided to separate these two concerns. We then applied domain engineering to identify the reusable elements on both sides. This led to two feature models, one for task requirement and the other for software components, enriched with intra- and inter-models constraints.

To manipulate these models, we are developing a generic feature diagram editor using ECLIPSE meta-modeling facilities (EMF, ECORE, GMF...). At this time, we have a first prototype that allowed us to represent both models. However, the current tool only supports natural language constraints, but we have experimented the KERMETA [1] workbench to implement some model to model transformations.

A concrete objective is to provide a graphic tool to assist the full specification and design of complete video surveillance processing chains. The tool should allow a designer to: (i) select the needed features from the task model, (ii) enforce validity constraints, (iii) automatically generate a specialized framework model corresponding to the specifications, (iv) instantiate this framework model under de-

signer's control to obtain the final processing chain, (v) automatically generate the glue code to put components together. Control during execution will be the matter of other techniques such as Program Supervision.

A number of scientific and technical issues are yet to be solved. First, we need to choose a formal way to represent internal constraints in feature models. Our intend is to rely on existing formalisms that provide manipulation and verification means. Second, after variability modeling, the next key point is to decide how to express the model to model transformations. This involves to define a model of transformations and a description language, to verify or ensure that transformations fulfill internal constraints, and to implement the transformations themselves. We also plan to use existing tools, preferably available within ECLIPSE.

In the long term, we may consider other representations, particularly for software components. Although feature models conveniently represent variability concerns, it would be suitable to make them compatible with standard component models [16]. Concerning model transformations, we wish to combine imperative and rule-based approaches, taking advantage of our previous work on Program Supervision that proposes a dynamic model of the framework [10]. It also includes all the work flow aspects necessary at execution time. Such a model could complement the static configuration obtained through model engineering, adding a run-time dimension. The artificial intelligence mechanisms that it involves could also complete dynamic adaptation approaches such as the DIVA one [11].

To conclude, our approach is based on a twofold separation. The first one distinguishes between the general video surveillance domain and particular applications, the second between task specification and component design. The first deals with variability, the second with model transformation. Although preliminary, our first results show that current model-driven engineering techniques can facilitate the specification and design of complex real life applications. In return, we expect that the confrontation may raise new issues to enrich the MDE paradigm.

References

[1] Kermeta home page. http://www.kermeta.org/.

[2] A. Avanzi, F. Brémond, C. Tornieri, and M. Thonnat. Design and assessment of an intelligent activity monitoring platform. *EURASIP Journal on Applied Signal Processing*, 14(8):2359–2374, 2005.

[3] D. S. Batory. Feature models, grammars, and propositional formulas. In J. H. Obbink and K. Pohl, editors, *SPLC*, volume 3714 of *LNCS*, pages 7–20. Springer, 2005.

[4] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[5] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged Configuration through Specialization and Multilevel Configuration of Feature Models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005.

[6] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Nov. 1990.

[7] K. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. Form: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5(1):143–168, 1998.

[8] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture–Practice and Promise*. Addison-Wesley , April 2003.

[9] C. W. Krueger. New methods in software product line development. In *10th Int. Software Product Line Conf.*, pages 95–102, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[10] S. Moisan. Knowledge representation for program reuse. In *Proc. European Conference of Artificial Intelligence*, Lyon, France, July 2002.

[11] B. Morin, F. Fleurey, N. Bencomo, J.-M. Jézéquel, A. Solberg, V. Delhen, and G. Blair. An aspect-oriented and model-driven approach for managing dynamic variability. In *Proc. ACM/IEEE 11th Int. Conf. MODELS'08*, volume 5301 of *LNCS*, pages 782–796, 2008.

[12] G. Perrouin, F. Chauvel, J. DeAntoni, and J.-M. Jézéquel. Modeling the variability space of self-adaptive applications. In S. Thiel and K. Pohl, editors, *2nd Dynamic Software Product Lines Workshop (SPLC 2008, Volume 2)*, pages 15–22, Limerick, Ireland, Sept. 2008. IEEE Computer Society.

[13] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[14] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51(2):456–479, 2007.

[15] M. Svahnberg, J. van Gurp, and J. Bosch. A taxonomy of variability realization techniques: Research articles. *Software Practice and Experience*, 35(8):705–754, 2005.

[16] C. Szyperski, D. Gruntz, and S. Murer. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley, 2002.

[17] M. Thonnat and S. Moisan. What can Program Supervision Do for Software Reuse? *IEE Proceedings - Software. Special Issue on Knowledge Modelling for Software Components Reuse*, 147(5):179–185, October 2000.

[18] J. White, D. C. Schmidt, E. Wuchner, and A. Nechypurenko. Automating product-line variant selection for mobile devices. In *SPLC*, pages 129–140. IEEE Computer Society, 2007.

[19] T. Ziadi and J.-M. Jézéquel. Software product line engineering with the UML: Deriving products. In T. Käkölä and J. C. Dueñas, editors, *Software Product Lines*, pages 557–588. Springer, 2006.